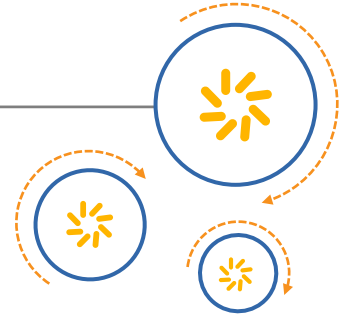


Exhibit LL

PUBLIC VERSION



Qualcomm Technologies, Inc.



Qualcomm[®] Hexagon[™] V66

Programmer's Reference Manual

80-N2040-42 A

November 17, 2017

Qualcomm Hexagon is a product of Qualcomm Technologies, Inc. Other Qualcomm products referenced herein are products of Qualcomm Technologies, Inc. or its subsidiaries.

Qualcomm, Hexagon, and HVX are trademarks of Qualcomm Incorporated, registered in the United States and other countries. Other product and brand names may be trademarks or registered trademarks of their respective owners.

This technical data may be subject to U.S. and international export, re-export, or transfer ("export") laws. Diversion contrary to U.S. and international law is strictly prohibited.

Qualcomm Technologies, Inc.
5775 Morehouse Drive
San Diego, CA 92121
U.S.A.

Contents

Figures	14
Tables	15
1 Introduction.....	17
1.1 Features.....	17
1.2 Functional units	19
1.2.1 Memory.....	20
1.2.2 Registers.....	20
1.2.3 Sequencer.....	20
1.2.4 Execution units.....	20
1.2.5 Load/store units.....	20
1.3 Instruction set	21
1.3.1 Addressing modes.....	21
1.3.2 Scalar operations.....	22
1.3.3 Vector operations	22
1.3.4 Floating-point operations	23
1.3.5 Program flow	24
1.3.6 Instruction packets	24
1.3.7 Dot-new instructions.....	25
1.3.8 Compound instructions	25
1.3.9 Duplex instructions	25
1.3.10 Instruction classes	25
1.3.11 Instruction intrinsics.....	26
1.4 Notation	27
1.4.1 Instruction syntax.....	27
1.4.2 Register operands.....	28
1.4.3 Numeric operands	30
1.5 Terminology	31
1.6 Technical assistance.....	31
2 Registers	32
2.1 General registers.....	33
2.2 Control registers	35
2.2.1 Program counter.....	37
2.2.2 Loop registers.....	38

2.2.3	User status register	38
2.2.4	Modifier registers	41
2.2.5	Predicate registers	42
2.2.6	Circular start registers	43
2.2.7	User general pointer register	43
2.2.8	Global pointer	43
2.2.9	Cycle count registers	44
2.2.10	Frame limit register	44
2.2.11	Frame key register	45
2.2.12	Packet count registers	45
2.2.13	Qtimer registers	46
3	Instructions	47
3.1	Instruction syntax	47
3.2	Instruction classes	49
3.3	Instruction packets	50
3.3.1	Packet execution semantics	51
3.3.2	Sequencing semantics	51
3.3.3	Resource constraints	52
3.3.4	Grouping constraints	53
3.3.5	Dependency constraints	54
3.3.6	Ordering constraints	54
3.3.7	Alignment constraints	55
3.4	Instruction intrinsics	55
3.5	Compound instructions	56
3.6	Duplex instructions	56
4	Data Processing	57
4.1	Data types	58
4.1.1	Fixed-point data	58
4.1.2	Floating-point data	58
4.1.3	Complex data	58
4.1.4	Vector data	58
4.2	Instruction options	60
4.2.1	Fractional scaling	60
4.2.2	Saturation	60
4.2.3	Arithmetic rounding	61
4.2.4	Convergent rounding	61
4.2.5	Scaling for divide and square-root	61
4.3	XTYPE operations	62
4.3.1	ALU	62
4.3.2	Bit manipulation	63

4.3.3	Complex.....	63
4.3.4	Floating point.....	64
4.3.5	Multiply.....	65
4.3.6	Permute.....	67
4.3.7	Predicate.....	67
4.3.8	Shift.....	68
4.4	ALU32 operations.....	69
4.5	Vector operations.....	70
4.6	CR operations.....	72
4.7	Compound operations.....	72
4.8	Special operations.....	72
4.8.1	H.264 CABAC processing.....	73
4.8.1.1	CABAC implementation.....	74
4.8.1.2	Code example.....	75
4.8.2	IP internet checksum.....	76
4.8.2.1	Code example.....	77
4.8.3	Software-defined radio.....	78
4.8.3.1	Rake despreading.....	78
4.8.3.2	Polynomial operations.....	79
5	Memory.....	81
5.1	Memory model.....	82
5.1.1	Address space.....	82
5.1.2	Byte order.....	82
5.1.3	Alignment.....	82
5.2	Memory loads.....	83
5.3	Memory stores.....	84
5.4	Dual stores.....	84
5.5	Slot 1 store with slot 0 load.....	85
5.6	New-value stores.....	85
5.7	Mem-ops.....	86
5.8	Addressing modes.....	86
5.8.1	Absolute.....	87
5.8.2	Absolute-set.....	87
5.8.3	Absolute with register offset.....	87
5.8.4	Global pointer relative.....	88
5.8.5	Indirect.....	89
5.8.6	Indirect with offset.....	89
5.8.7	Indirect with register offset.....	89
5.8.8	Indirect with auto-increment immediate.....	90
5.8.9	Indirect with auto-increment register.....	90
5.8.10	Circular with auto-increment immediate.....	91
5.8.11	Circular with auto-increment register.....	93

5.8.12 Bit-reversed with auto-increment register	94
5.9 Conditional load/stores	95
5.10 Cache memory	96
5.10.1 Uncached memory	97
5.10.2 Tightly coupled memory	97
5.10.3 Cache maintenance operations	97
5.10.4 L2 cache operations	98
5.10.5 Cache line zero	99
5.10.6 Cache prefetch	99
5.11 Memory ordering	102
5.12 Atomic operations	103
6 Conditional Execution.....	105
6.1 Scalar predicates	105
6.1.1 Generating scalar predicates	106
6.1.2 Consuming scalar predicates	108
6.1.3 Auto-AND predicates	109
6.1.4 Dot-new predicates	110
6.1.5 Dependency constraints	111
6.2 Vector predicates	111
6.2.1 Vector compare	111
6.2.2 Vector mux instruction	113
6.2.3 Using vector conditionals	114
6.3 Predicate operations	115
7 Program Flow.....	116
7.1 Conditional instructions	116
7.2 Hardware loops	117
7.2.1 Loop setup	118
7.2.2 Loop end	119
7.2.3 Loop execution	120
7.2.4 Pipelined hardware loops	121
7.2.5 Loop restrictions	124
7.3 Software branches	124
7.3.1 Jumps	125
7.3.2 Calls	125
7.3.3 Returns	126
7.3.4 Extended branches	127
7.3.5 Branches to and from packets	127
7.4 Speculative jumps	128
7.5 Compare jumps	129
7.5.1 New-value compare jumps	130

7.6 Register transfer jumps.....	131
7.7 Dual jumps.....	132
7.8 Hint indirect jump target	133
7.9 Pauses	134
7.10 Exceptions	134
8 Software Stack	137
8.1 Stack structure	137
8.2 Stack frames	139
8.3 Stack protection	139
8.3.1 Stack bounds checking.....	139
8.3.2 Stack smashing protection	140
8.4 Stack registers.....	140
8.5 Stack instructions.....	141
9 PMU Events	142
9.1 V66 processor event symbols.....	143
10 Instruction Encoding.....	153
10.1 Instructions	153
10.2 Sub-instructions.....	154
10.3 Duplexes	157
10.4 Instruction classes.....	159
10.5 Instruction packets.....	160
10.6 Loop packets.....	161
10.7 Immediate values.....	162
10.8 Scaled immediates	162
10.9 Constant extenders.....	163
10.10 New-value operands	166
10.11 Instruction mapping.....	167
11 Instruction Set.....	168
11.1 ALU32	171
11.1.1 ALU32/ALU	171
Logical operations	173
Negate.....	175
Nop	176
Subtract.....	177
Sign extend	179
Transfer immediate.....	180
Transfer register	182
Vector add halfwords	183

Vector average halfwords.....	184
Vector subtract halfwords	185
Zero extend.....	187
11.1.2 ALU32/PERM	188
Mux	191
Shift word by 16	193
Pack high and low halfwords	195
11.1.3 ALU32/PRED	196
Conditional shift halfword.....	198
Conditional combine	200
Conditional logical operations.....	201
Conditional subtract	203
Conditional sign extend.....	204
Conditional transfer.....	206
Conditional zero extend.....	207
Compare	209
Compare to general register	211
11.2 CR.....	212
Corner detection acceleration.....	214
Logical reductions on predicates.....	215
Looping instructions.....	216
Add to PC	218
Pipelined loop instructions	219
Logical operations on predicates.....	221
User control register transfer	223
11.3 JR.....	224
Hint an indirect jump address.....	226
Jump to address from register	227
11.4 J.....	228
Compare and jump	230
Jump to address	234
Jump to address conditioned on new predicate	236
Jump to address condition on register value	237
Transfer and jump	239
11.5 LD.....	240
Load doubleword conditionally.....	243
Load byte.....	245
Load byte conditionally.....	247
Load byte into shifted vector.....	249
Load half into shifted vector	252
Load halfword	255
Load halfword conditionally	257
Load unsigned byte	259

Load unsigned byte conditionally	261
Load unsigned halfword	263
Load unsigned halfword conditionally	265
Load word	267
Load word conditionally	269
Deallocate stack frame	271
Deallocate frame and return	273
Load and unpack bytes to halfwords	275
11.6 MEMOP	283
Operation on memory halfword	285
Operation on memory word	286
11.7 NV	287
11.7.1 NV/J	287
11.7.2 NV/ST	291
Store new-value byte conditionally	293
Store new-value halfword	296
Store new-value halfword conditionally	298
Store new-value word	301
Store new-value word conditionally	303
11.8 ST	306
Store doubleword conditionally	309
Store byte	311
Store byte conditionally	313
Store halfword	316
Store halfword conditionally	319
Store word	322
Store word conditionally	324
Allocate stack frame	327
11.9 SYSTEM	329
11.9.1 SYSTEM/GUEST	329
11.9.2 SYSTEM/MONITOR	331
Swap SGP control register	332
Cancel pending interrupts	333
Data cache kill	334
Data cache maintenance monitor instructions	335
Read the interrupt mask for a thread	337
Acquire hardware lock	338
Release hardware lock	339
Interrupt to thread assignment read	340
Interrupt to thread assignment write	342
Instruction cache maintenance supervisor operations	344
Instruction cache maintenance operations (single-thread)	345
L2 cache operations by index	346

L2 cache global operations.....	347
L2 cache operations by address.....	349
L2 tag read/write	351
Load from physical address.....	353
Raise NMI on threads.....	354
Resume from Wait mode.....	355
Return from exception.....	356
Set the interrupt mask for a thread	357
Set the priority for a thread.....	358
Set interrupt auto disable	359
Start threads	360
Stop threads	361
Software interrupt.....	362
TLB read/write/probe operations	363
System control register transfer.....	365
Transition threads to Wait mode	367
11.9.3 SYSTEM/USER.....	368
Store conditional.....	369
Zero a cache line.....	371
Memory barrier.....	372
Breakpoint	373
Data cache prefetch	374
Data cache maintenance user operations.....	375
Instruction cache maintenance user operations	377
Instruction synchronization	378
L2 cache prefetch	379
Pause.....	382
Memory thread synchronization.....	383
Send value to ETM trace	384
Trap	385
11.10 XTYPE	387
11.10.1 XTYPE/ALU	387
Absolute value word.....	388
Add and accumulate	389
Add doublewords	391
Add halfword.....	393
Add or subtract doublewords with carry	395
Clip to unsigned.....	396
Logical doublewords	397
Logical-logical doublewords.....	399
Logical-logical words.....	400
Maximum words.....	402
Maximum doublewords.....	403

Minimum words	404
Minimum doublewords	405
Modulo wrap	406
Negate	407
Round	408
Subtract doublewords	411
Subtract and accumulate words	412
Subtract halfword	413
Sign extend word to doubleword	415
Vector absolute value halfwords	416
Vector absolute value words	417
Vector absolute difference bytes	418
Vector absolute difference halfwords	419
Vector absolute difference words	420
Vector add compare and select maximum bytes	421
Vector add compare and select maximum halfwords	422
Vector add halfwords	424
Vector add halfwords with saturate and pack to unsigned bytes	426
Vector reduce add unsigned bytes	427
Vector reduce add halfwords	429
Vector add bytes	431
Vector add words	432
Vector average halfwords	433
Vector average unsigned bytes	435
Vector average words	436
Vector clip to unsigned	438
Vector conditional negate	439
Vector maximum bytes	441
Vector maximum halfwords	442
Vector reduce maximum halfwords	443
Vector reduce maximum words	445
Vector maximum words	447
Vector minimum bytes	448
Vector minimum halfwords	450
Vector reduce minimum halfwords	451
Vector reduce minimum words	453
Vector minimum words	455
Vector sum of absolute differences unsigned bytes	456
Vector subtract halfwords	458
Vector subtract bytes	460
Vector subtract words	461
11.10.2 XTYPE/BIT	462
Count population	464

Count trailing.....	465
Extract bitfield.....	466
Insert bitfield	469
Interleave/deinterleave	471
Linear feedback-shift iteration	472
Masked parity	473
Bit reverse.....	474
Set/clear/toggle bit.....	475
Split bitfield	477
Table index.....	479
11.10.3 XTYPE/COMPLEX.....	482
Complex add/sub words	485
Complex multiply.....	487
Complex multiply real or imaginary	490
Complex multiply with round and pack	492
Complex multiply 32x16.....	494
Complex multiply real or imaginary 32-bit.....	496
Vector complex multiply real or imaginary	500
Vector complex conjugate.....	503
Vector complex rotate	504
Vector reduce complex multiply real or imaginary.....	506
Vector reduce complex multiply by scalar.....	509
Vector reduce complex multiply by scalar with round and pack	512
Vector reduce complex rotate.....	514
11.10.4 XTYPE/FP	517
Floating point addition	517
Classify floating-point value	518
Compare floating-point value.....	519
Convert floating-point value to other format	521
Convert integer to floating-point value	522
Convert floating-point value to integer	524
Floating point extreme value assistance.....	526
Floating point fused multiply-add	527
Floating point fused multiply-add with scaling.....	528
Floating point reciprocal square root approximation	529
Floating point fused multiply-add for library routines.....	530
Create floating-point constant	532
Floating point maximum	533
Floating point minimum.....	534
Floating point multiply	535
Floating point reciprocal approximation	536
Floating point subtraction.....	537
11.10.5 XTYPE/MPY	538

Vector multiply word by signed half (32x16)	541
Vector multiply word by unsigned half (32x16)	545
Multiply signed halfwords	549
Multiply unsigned halfwords	556
Polynomial multiply words	561
Vector reduce multiply word by signed half (32x16)	563
Multiply and use upper result	565
Multiply and use full result	568
Vector dual multiply	570
Vector dual multiply with round and pack	573
Vector reduce multiply bytes	575
Vector dual multiply signed by unsigned bytes	577
Vector multiply even halfwords	579
Vector multiply halfwords	581
Vector multiply halfwords with round and pack	583
Vector multiply halfwords, signed by unsigned	585
Vector reduce multiply halfwords	587
Vector multiply bytes	589
Vector polynomial multiply halfwords	591
11.10.6 XTYPE/PERM	593
Saturate	595
Swizzle bytes	597
Vector align	598
Vector round and pack	600
Vector saturate and pack	602
Vector saturate without pack	605
Vector shuffle	607
Vector splat bytes	609
Vector splat halfwords	610
Vector splice	611
Vector sign extend	612
Vector truncate	614
Vector zero extend	616
11.10.7 XTYPE/PRED	618
Compare byte	620
Compare half	622
Compare doublewords	624
Compare bit mask	625
Mask generate from predicate	626
Check for TLB match	627
Predicate transfer	628
Test bit	629
Vector compare halfwords	630

Vector compare bytes for any match.....	632
Vector compare bytes	633
Vector compare words.....	635
Viterbi pack even and odd predicate bits	637
Vector mux	638
11.10.8 XTYPE/SHIFT.....	639
Shift by immediate	640
Shift by immediate and accumulate	642
Shift by immediate and add.....	645
Shift by immediate and logical.....	646
Shift right by immediate with rounding	650
Shift left by immediate with saturation	652
Shift by register	653
Shift by register and accumulate	656
Shift by register and logical.....	659
Shift by register with saturation	662
Vector shift halfwords by immediate	664
Vector arithmetic shift halfwords with round	666
Vector arithmetic shift halfwords with saturate and pack	668
Vector shift halfwords by register.....	670
Vector shift words by immediate	672
Vector shift words by register	674
Vector shift words with truncate and pack	676
Instruction Index	678
Intrinsics Index	693

Figures

Figure 1-1	Hexagon V66 processor architecture	19
Figure 1-2	Vector instruction example	23
Figure 1-3	Instruction classes and combinations	26
Figure 1-4	Register field symbols	29
Figure 2-1	General registers	33
Figure 2-2	Control registers	35
Figure 3-1	Packet grouping combinations	52
Figure 4-1	Vector byte operation	59
Figure 4-2	Vector halfword operation	59
Figure 4-3	Vector word operation	60
Figure 4-4	64-bit shift and add/sub/logical	68
Figure 4-5	Vector halfword shift right	71
Figure 5-1	Hexagon processor byte order	82
Figure 5-2	L2FETCH instruction	101
Figure 6-1	Vector byte compare	112
Figure 6-2	Vector halfword compare	112
Figure 6-3	Vector mux instruction	113
Figure 8-1	Stack structure	138
Figure 10-1	Instruction packet encoding	160

Tables

Table 1-1	Register symbols	28
Table 1-2	Register bit field symbols	29
Table 1-3	Instruction operands	30
Table 1-4	Data symbols	31
Table 2-1	General register aliases	34
Table 2-2	General register pairs	34
Table 2-3	Aliased control registers	36
Table 2-4	Control register pairs	37
Table 2-5	Loop registers	38
Table 2-6	User status register	39
Table 2-7	Modifier registers (indirect auto-increment addressing)	41
Table 2-8	Modifier registers (circular addressing)	41
Table 2-9	Modifier registers (bit-reversed addressing)	42
Table 2-10	Predicate registers	42
Table 2-11	Circular start registers	43
Table 2-12	User general pointer register	43
Table 2-13	Global pointer register	43
Table 2-14	Cycle count registers	44
Table 2-15	Frame limit register	44
Table 2-16	Frame key register	45
Table 2-17	Packet count registers	45
Table 2-18	Qtimer registers	46
Table 3-1	Instruction symbols	47
Table 3-2	Instruction classes	49
Table 4-1	Single-precision multiply options	66
Table 4-2	Double precision multiply options	66
Table 4-3	Control register transfer instructions	72
Table 5-1	Memory alignment restrictions	83
Table 5-2	Load instructions	83
Table 5-3	Store instructions	84
Table 5-4	Mem-ops	86
Table 5-5	Addressing modes	86
Table 5-6	Offset ranges (global pointer relative)	88
Table 5-7	Offset ranges (Indirect with offset)	89
Table 5-8	Increment ranges (Indirect with auto-inc immediate)	90
Table 5-9	Increment ranges (Circular with auto-inc immediate)	91
Table 5-10	Increment ranges (Circular with auto-inc register)	93
Table 5-11	Addressing modes (Conditional load/store)	95
Table 5-12	Conditional offset ranges (Indirect with offset)	96
Table 5-13	Cache instructions (User-level)	98
Table 5-14	Memory ordering instructions	102

Table 5-15	Atomic instructions	103
Table 6-1	Scalar predicate-generating instructions	106
Table 6-2	Vector mux instruction	113
Table 6-3	Predicate register instructions	115
Table 7-1	Loop instructions	118
Table 7-2	Software pipelined loop	122
Table 7-3	Software pipelined loop (using spNloop0)	123
Table 7-4	Software branch instructions	124
Table 7-5	Jump instructions	125
Table 7-6	Call instructions	125
Table 7-7	Return instructions	126
Table 7-8	Speculative jump instructions	128
Table 7-9	Compare jump instructions	130
Table 7-10	New-value compare jump instructions	131
Table 7-11	Register transfer jump instructions	132
Table 7-12	Dual jump instructions	132
Table 7-13	Jump hint instruction	133
Table 7-14	Pause instruction	134
Table 7-15	V66 exceptions	135
Table 8-1	Stack registers	140
Table 8-2	Stack instructions	141
Table 9-1	V66 processor events symbols	143
Table 10-1	Instruction fields	153
Table 10-2	Sub-instructions	155
Table 10-3	Sub-instruction registers	156
Table 10-4	Duplex instruction	157
Table 10-5	Duplex ICLASS field	157
Table 10-6	Instruction class encoding	159
Table 10-7	Loop packet encoding	161
Table 10-8	Scaled immediate encoding (indirect offsets)	162
Table 10-9	Constant extender encoding	163
Table 10-10	Constant extender instructions	164
Table 10-11	Instruction mapping	167
Table 11-1	Instruction syntax symbols	169
Table 11-2	Instruction operand symbols	170
Table 11-3	Instruction behavior symbols	170

1 Introduction

The Qualcomm Hexagon™ processor is a general-purpose digital signal processor designed for high performance and low power across a wide variety of multimedia and modem applications. V66 is a member of the sixth generation of the Hexagon processor architecture.

1.1 Features

■ Memory

Program code and data are stored in a unified 32-bit address space. The load/store architecture supports a complete set of addressing modes for both compiler code generation and DSP application programming.

■ Registers

Thirty two 32-bit general purpose registers can be accessed as single registers or as 64-bit register pairs. The general registers hold all data including scalar, pointer, and packed vector data.

■ Data types

Instructions can perform a wide variety of operations on fixed-point or floating-point data. The fixed-point operations support scalar and vector data in a variety of sizes. The floating-point operations support single-precision data.

■ Parallel execution

Instructions can be grouped into very long instruction word (VLIW) packets for parallel execution, with each packet containing from one to four instructions. Vector instructions operate on single instruction multiple data (SIMD) vectors.

■ Program flow

Nestable zero-overhead hardware loops are supported. Conditional/unconditional jumps and subroutine calls support both PC-relative and register indirect addressing. Two program flow instructions can be grouped into one packet.

■ Instruction pipeline

Pipeline hazards are resolved by the hardware: instruction scheduling is not constrained by pipeline restrictions.

■ Code compression

Compound instructions merge certain common operation sequences (add-accumulate, shift-add, etc.) into a single instruction. *Duplex* encodings express two parallel instructions in a single 32-bit word.

■ **Cache memory**

Memory accesses can be cached or uncached. Separate L1 instruction and data caches exist for program code and data. A unified L2 cache can be partly or wholly configured as tightly-coupled memory (TCM).

■ **Virtual memory**

Memory is addressed virtually, with virtual-to-physical memory mapping handled by a resident OS. Virtual memory supports the implementation of memory management and memory protection in a hardware-independent manner.

Figure 1-3 presents an overview of the instruction classes and how they can be grouped together.

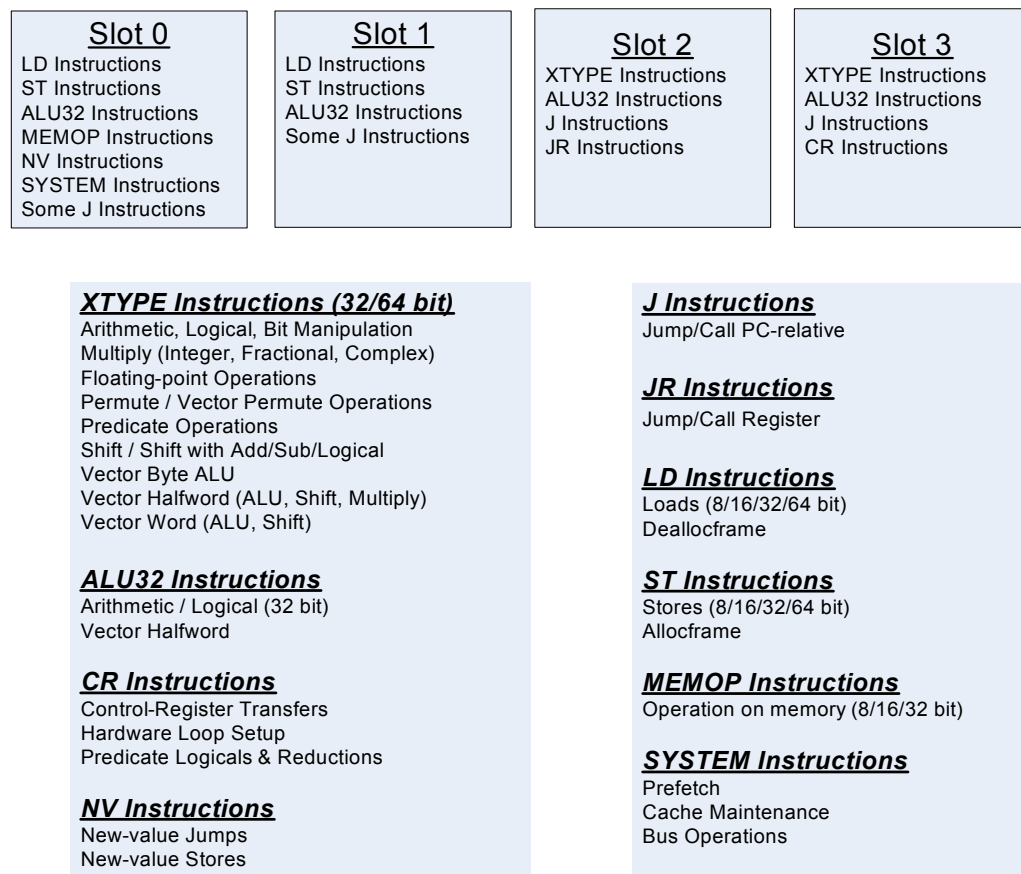


Figure 1-3 Instruction classes and combinations

1.3.11 Instruction intrinsics

To support efficient coding of the time-critical sections of a program (without resorting to assembly language), the C compilers support intrinsics which are used to directly express Hexagon processor instructions from within C code. For example:

```
int main()
{
    long long v1 = 0xFFFF0000FFFF0000;
    long long v2 = 0x0000FFFF0000FFFF;
    long long result;

    // find the minimum for each half-word in 64-bit vector
    result = Q6_P_vminh_PP(v1,v2);
}
```

Intrinsics are defined for most of the Hexagon processor instructions.

1.4 Notation

This section presents the notational conventions used in this document to describe Hexagon processor instructions:

- Instruction syntax
- Register operands
- Numeric operands

NOTE The notation described here does not appear in actual assembly language instructions. It is used only to specify the instruction syntax and behavior.

1.4.1 Instruction syntax

The following notation is used to describe the syntax of instructions:

- Monospaced font is used for instructions
- Square brackets enclose optional items (e.g., `[:sat]`, means that saturation is optional)
- Braces indicate a choice of items (e.g., `{Rs, #s16}`, means that either Rs or a signed 16-bit immediate can be used)

11.10 XTYPE

The XTYPE instruction class includes instructions which perform most of the data processing done by the Hexagon processor.

XTYPE instructions are executable on slot 2 or slot 3.

11.10.1 XTYPE/ALU

The XTYPE/ALU instruction subclass includes instructions which perform arithmetic and logical operations.

Absolute value doubleword

Take the absolute value of the 64-bit source register and place it in the destination register.

Syntax

`Rdd=abs (Rss)`

Behavior

`Rdd = ABS (Rss) ;`

Class: XTYPE (slots 2,3)

Intrinsics

`Rdd=abs (Rss)`

`Word64 Q6_P_abs_P (Word64 Rss)`

Encoding

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	
ICLASS				RegType				MajOp			s5					Parse								MinOp			d5					
1	0	0	0	0	0	0	0	1	0	0	s	s	s	s	s	P	P	-	-	-	-	-	-	1	1	0	d	d	d	d	d	Rdd=abs(Rss)

Field name	Description
ICLASS	Instruction Class
Parse	Packet/Loop parse bits
d5	Field to encode register d
s5	Field to encode register s
MajOp	Major Opcode
MinOp	Minor Opcode
RegType	Register Type

11.10.2 XTYPE/BIT

The XTYPE/BIT instruction subclass includes instructions for bit manipulation.

Count leading

Count leading zeros (cl0) counts the number of consecutive zeros starting with the most significant bit.

Count leading ones (cl1) counts the number of consecutive ones starting with the most significant bit.

Count leading bits (clb) counts both leading ones and leading zeros and then selects the maximum.

The NORMAMT instruction returns the number of leading bits minus one.

For a two's-complement number, the number of leading zeros is zero for negative numbers. The number of leading ones is zero for positive numbers.

The number of leading bits can be used to judge the magnitude of the value.

Syntax	Behavior
<code>Rd=add(clb(Rs),#s6)</code>	<code>Rd = (max(count_leading_ones(Rs),count_leading_ones(~Rs)))+#s;</code>
<code>Rd=add(clb(Rss),#s6)</code>	<code>Rd = (max(count_leading_ones(Rss),count_leading_ones(~Rss)))+#s;</code>
<code>Rd=cl0(Rs)</code>	<code>Rd = count_leading_ones(~Rs);</code>
<code>Rd=cl0(Rss)</code>	<code>Rd = count_leading_ones(~Rss);</code>
<code>Rd=cl1(Rs)</code>	<code>Rd = count_leading_ones(Rs);</code>
<code>Rd=cl1(Rss)</code>	<code>Rd = count_leading_ones(Rss);</code>
<code>Rd=clb(Rs)</code>	<code>Rd = max(count_leading_ones(Rs),count_leading_ones(~Rs));</code>
<code>Rd=clb(Rss)</code>	<code>Rd = max(count_leading_ones(Rss),count_leading_ones(~Rss));</code>
<code>Rd=normamt(Rs)</code>	<pre>if (Rs == 0) { Rd = 0; } else { Rd = (max(count_leading_ones(Rs),count_leading_ones(~Rs)))-1; }</pre>
<code>Rd=normamt(Rss)</code>	<pre>if (Rss == 0) { Rd = 0; } else { Rd = (max(count_leading_ones(Rss),count_leading_ones(~Rss)))-1; }</pre>

Class: XTYPE (slots 2,3)**Intrinsics**

Rd=add(clb(Rs),#s6)	Word32 Q6_R_add_clb_RI(Word32 Rs, Word32 Is6)
Rd=add(clb(Rss),#s6)	Word32 Q6_R_add_clb_PI(Word64 Rss, Word32 Is6)
Rd=cl0(Rs)	Word32 Q6_R_cl0_R(Word32 Rs)
Rd=cl0(Rss)	Word32 Q6_R_cl0_P(Word64 Rss)
Rd=cl1(Rs)	Word32 Q6_R_cl1_R(Word32 Rs)
Rd=cl1(Rss)	Word32 Q6_R_cl1_P(Word64 Rss)
Rd=clb(Rs)	Word32 Q6_R_clb_R(Word32 Rs)
Rd=clb(Rss)	Word32 Q6_R_clb_P(Word64 Rss)
Rd=normamt(Rs)	Word32 Q6_R_normamt_R(Word32 Rs)
Rd=normamt(Rss)	Word32 Q6_R_normamt_P(Word64 Rss)

Encoding

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	
ICLASS				RegType				MajOp				s5				Parse								MinOp				d5				
1	0	0	0	1	0	0	0	0	1	0	s	s	s	s	s	P	P	-	-	-	-	-	-	0	0	0	d	d	d	d	d	Rd=clb(Rss)
1	0	0	0	1	0	0	0	0	1	0	s	s	s	s	s	P	P	-	-	-	-	-	-	0	1	0	d	d	d	d	d	Rd=cl0(Rss)
1	0	0	0	1	0	0	0	0	1	0	s	s	s	s	s	P	P	-	-	-	-	-	-	1	0	0	d	d	d	d	d	Rd=cl1(Rss)
1	0	0	0	1	0	0	0	0	1	1	s	s	s	s	s	P	P	-	-	-	-	-	-	0	0	0	d	d	d	d	d	Rd=normamt(Rss)
1	0	0	0	1	0	0	0	0	1	1	s	s	s	s	s	P	P	i	i	i	i	i	i	0	1	0	d	d	d	d	d	Rd=add(clb(Rss),#s6)
1	0	0	0	1	1	0	0	0	0	1	s	s	s	s	s	P	P	i	i	i	i	i	i	0	0	0	d	d	d	d	d	Rd=add(clb(Rs),#s6)
1	0	0	0	1	1	0	0	0	0	0	s	s	s	s	s	P	P	-	-	-	-	-	-	1	0	0	d	d	d	d	d	Rd=clb(Rs)
1	0	0	0	1	1	0	0	0	0	0	s	s	s	s	s	P	P	-	-	-	-	-	-	1	0	1	d	d	d	d	d	Rd=cl0(Rs)
1	0	0	0	1	1	0	0	0	0	0	s	s	s	s	s	P	P	-	-	-	-	-	-	1	1	0	d	d	d	d	d	Rd=cl1(Rs)
1	0	0	0	1	1	0	0	0	0	0	s	s	s	s	s	P	P	-	-	-	-	-	-	1	1	1	d	d	d	d	d	Rd=normamt(Rs)

Field name	Description
ICLASS	Instruction Class
Parse	Packet/Loop parse bits
d5	Field to encode register d
s5	Field to encode register s
MajOp	Major Opcode
MinOp	Minor Opcode
RegType	Register Type

Count population

Population Count (popcount) counts the number of bits in Rss that are set.

Syntax

```
Rd=popcount (Rss)
```

Behavior

```
Rd = count_ones (Rss) ;
```

Class: XTYPE (slots 2,3)

Intrinsics

```
Rd=popcount (Rss)
```

```
Word32 Q6_R_popcount_P (Word64 Rss)
```

Encoding

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	
ICLASS				RegType				MajOp		s5					Parse		MinOp							d5								
1	0	0	0	1	0	0	0	0	1	1	s	s	s	s	s	P	P	-	-	-	-	-	-	0	1	1	d	d	d	d	d	Rd=popcount(Rss)

Field name	Description
ICLASS	Instruction Class
Parse	Packet/Loop parse bits
d5	Field to encode register d
s5	Field to encode register s
MajOp	Major Opcode
MinOp	Minor Opcode
RegType	Register Type

Count trailing

Count trailing zeros (ct0) counts the number of consecutive zeros starting with the least significant bit.

Count trailing ones (ct1) counts the number of consecutive ones starting with the least significant bit.

Syntax

`Rd=ct0(Rs)`

`Rd=ct0(Rss)`

`Rd=ct1(Rs)`

`Rd=ct1(Rss)`

Behavior

`Rd = count_leading_ones(~reverse_bits(Rs));`

`Rd = count_leading_ones(~reverse_bits(Rss));`

`Rd = count_leading_ones(reverse_bits(Rs));`

`Rd = count_leading_ones(reverse_bits(Rss));`

Class: XTYPE (slots 2,3)

Intrinsics

`Rd=ct0(Rs)`

`Rd=ct0(Rss)`

`Rd=ct1(Rs)`

`Rd=ct1(Rss)`

`Word32 Q6_R_ct0_R(Word32 Rs)`

`Word32 Q6_R_ct0_P(Word64 Rss)`

`Word32 Q6_R_ct1_R(Word32 Rs)`

`Word32 Q6_R_ct1_P(Word64 Rss)`

Encoding

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	
ICLASS				RegType				MajOp				s5				Parse								MinOp				d5				
1	0	0	0	1	0	0	0	1	1	1	s	s	s	s	s	P	P	-	-	-	-	-	-	0	1	0	d	d	d	d	d	Rd=ct0(Rss)
1	0	0	0	1	0	0	0	1	1	1	s	s	s	s	s	P	P	-	-	-	-	-	-	1	0	0	d	d	d	d	d	Rd=ct1(Rss)
1	0	0	0	1	1	0	0	0	1	0	s	s	s	s	s	P	P	-	-	-	-	-	-	1	0	0	d	d	d	d	d	Rd=ct0(Rs)
1	0	0	0	1	1	0	0	0	1	0	s	s	s	s	s	P	P	-	-	-	-	-	-	1	0	1	d	d	d	d	d	Rd=ct1(Rs)

Field name

ICLASS

Parse

d5

s5

MajOp

MinOp

RegType

Description

Instruction Class

Packet/Loop parse bits

Field to encode register d

Field to encode register s

Major Opcode

Minor Opcode

Register Type

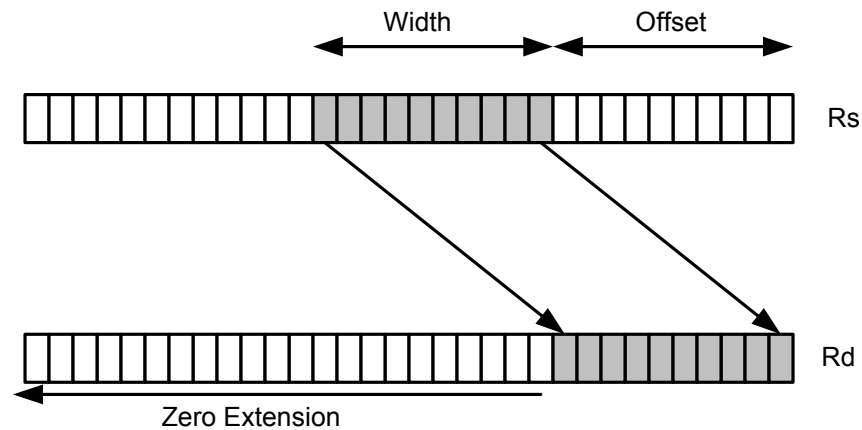
Extract bitfield

Extract a bitfield from the source register (or register pair) and deposit into the least significant bits of the destination register (or register pair). The other, more significant bits in the destination are either cleared or sign-extended, depending on the instruction.

The width of the extracted field is obtained from the first immediate or from the most-significant word of Rtt. The field offset is obtained from either the second immediate or from the least-significant word of Rtt.

For register-based extract, where Rtt supplies the offset and width, the offset value is treated as a signed 7-bit number. If this value is negative, the source register Rss is shifted left (the reverse direction). Width number of bits are then taken from the least-significant portion of this result.

If the shift amount and/or offset captures data beyond the most significant end of the input, these bits are taken as zero.



Syntax

`Rd=extract (Rs, #u5, #U5)`

`Rd=extract (Rs, Rtt)`

`Rd=extractu (Rs, #u5, #U5)`

`Rd=extractu (Rs, Rtt)`

`Rdd=extract (Rss, #u6, #U6)`

Behavior

```
width=#u;
offset=#U;
Rd = sxtwidth->32((Rs >> offset));
```

```
width=zxt6->32((Rtt.w[1]));
offset=sxt7->32((Rtt.w[0]));
Rd = sxtwidth->64((offset>0)?(zxt32->64(zxt32->64(Rs))>>>offset):(zxt32->64(zxt32->64(Rs))<<offset));
```

```
width=#u;
offset=#U;
Rd = zxtwidth->32((Rs >> offset));
```

```
width=zxt6->32((Rtt.w[1]));
offset=sxt7->32((Rtt.w[0]));
Rd = zxtwidth->64((offset>0)?(zxt32->64(zxt32->64(Rs))>>>offset):(zxt32->64(zxt32->64(Rs))<<offset));
```

```
width=#u;
offset=#U;
Rdd = sxtwidth->64((Rss >> offset));
```

Syntax**Behavior**

Rdd=extract (Rss, Rtt)

```
width=zxt6-32((Rtt.w[1]));
offset=sxt7-32((Rtt.w[0]));
Rdd = sxtwidth-
>64((offset>0)?(Rss>>offset):(Rss<<offset));
```

Rdd=extractu (Rss, #u6, #U6)

```
width=#u;
offset=#U;
Rdd = zxtwidth->64((Rss >> offset));
```

Rdd=extractu (Rss, Rtt)

```
width=zxt6-32((Rtt.w[1]));
offset=sxt7-32((Rtt.w[0]));
Rdd = zxtwidth-
>64((offset>0)?(Rss>>offset):(Rss<<offset));
```

Class: XTYPE (slots 2,3)**Intrinsics**

Rd=extract (Rs, #u5, #U5)

Word32 Q6_R_extract_RII (Word32 Rs, Word32 Iu5, Word32 IU5)

Rd=extract (Rs, Rtt)

Word32 Q6_R_extract_RP (Word32 Rs, Word64 Rtt)

Rd=extractu (Rs, #u5, #U5)

Word32 Q6_R_extractu_RII (Word32 Rs, Word32 Iu5, Word32 IU5)

Rd=extractu (Rs, Rtt)

Word32 Q6_R_extractu_RP (Word32 Rs, Word64 Rtt)

Rdd=extract (Rss, #u6, #U6)

Word64 Q6_P_extract_PII (Word64 Rss, Word32 Iu6, Word32 IU6)

Rdd=extract (Rss, Rtt)

Word64 Q6_P_extract_PP (Word64 Rss, Word64 Rtt)

Rdd=extractu (Rss, #u6, #U6)

Word64 Q6_P_extractu_PII (Word64 Rss, Word32 Iu6, Word32 IU6)

Rdd=extractu (Rss, Rtt)

Word64 Q6_P_extractu_PP (Word64 Rss, Word64 Rtt)

Encoding

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0			
ICLASS				RegType				MajOp			s5					Parse			MinOp							d5								
1	0	0	0	0	0	0	1	I	I	I	s	s	s	s	s	P	P	i	i	i	i	i	i	I	I	I	d	d	d	d	d	Rdd=extractu(Rss,#u6,#U6)		
1	0	0	0	1	0	1	0	I	I	I	s	s	s	s	s	P	P	i	i	i	i	i	i	I	I	I	d	d	d	d	d	Rdd=extract(Rss,#u6,#U6)		
1	0	0	0	1	1	0	1	0	I	I	s	s	s	s	s	P	P	0	i	i	i	i	i	I	I	I	d	d	d	d	d	Rd=extractu(Rs,#u5,#U5)		
1	0	0	0	1	1	0	1	1	I	I	s	s	s	s	s	P	P	0	i	i	i	i	i	I	I	I	d	d	d	d	d	Rd=extract(Rs,#u5,#U5)		
ICLASS				RegType				Maj			s5					Parse			t5							Min			d5					
1	1	0	0	0	0	0	1	0	0	-	s	s	s	s	s	P	P	-	t	t	t	t	t	0	0	-	d	d	d	d	d	Rdd=extractu(Rss,Rtt)		
1	1	0	0	0	0	0	1	1	1	-	s	s	s	s	s	P	P	-	t	t	t	t	t	1	0	-	d	d	d	d	d	Rdd=extract(Rss,Rtt)		
1	1	0	0	1	0	0	1	0	0	-	s	s	s	s	s	P	P	-	t	t	t	t	t	0	0	-	d	d	d	d	d	Rd=extractu(Rs,Rtt)		
1	1	0	0	1	0	0	1	0	0	-	s	s	s	s	s	P	P	-	t	t	t	t	t	0	1	-	d	d	d	d	d	Rd=extract(Rs,Rtt)		

Field name	Description
IClass	Instruction Class
Parse	Packet/Loop parse bits
d5	Field to encode register d
s5	Field to encode register s
t5	Field to encode register t
MajOp	Major Opcode
MinOp	Minor Opcode
Maj	Major Opcode
Min	Minor Opcode
RegType	Register Type
RegType	Register Type